



**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

BRNO UNIVERSITY OF TECHNOLOGY

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**

FACULTY OF INFORMATION TECHNOLOGY

**ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ**

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

**GRAFICKÉ INTRO 64KB S POUŽITÍM OPENGL**

GRAPHICS INTRO 64KB USING OPENGL

**BAKALÁŘSKÁ PRÁCE**

BACHELOR'S THESIS

**AUTOR PRÁCE**

AUTHOR

**JAKUB MENŠÍK**

**VEDOUCÍ PRÁCE**

SUPERVISOR

**Ing. MICHAL MATÝŠEK**

**BRNO 2018**

**Vysoké učení technické v Brně - Fakulta informačních technologií**

Ústav počítačové grafiky a multimédií

Akademický rok 2017/2018

**Zadání bakalářské práce**

Řešitel: **Menšík Jakub**

Obor: Informační technologie

Téma: **Grafické intro 64kB s použitím OpenGL**  
**Graphics Intro 64kB Using OpenGL**

Kategorie: Počítačová grafika

**Pokyny:**

1. Seznamte se s fenoménem grafického intra s omezenou velikostí.
2. Prostudujte knihovnu OpenGL a její nadstavby.
3. Popište vybrané techniky použitelné v grafickém intru s omezenou velikostí.
4. Implementujte grafické intro s použitím OpenGL, aby velikost spustitelné verze nepřesáhla 64kB.
5. Zhodnoťte dosažené výsledky a navrhněte možnosti pokračování projektu; vytvořte plakátek pro prezentování projektu.

**Literatura:**

- dle pokynů vedoucího

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese  
<http://www.fit.vutbr.cz/info/szz/>

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Matýšek Michal, Ing.**, UPGM FIT VUT

Datum zadání: 1. listopadu 2017

Datum odevzdání: 16. května 2018

**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**  
Fakulta informačních technologií  
Ústav počítačové grafiky a multimédií  
L. S. 602 00 Brno, Božetěchova 2



---

doc. Dr. Ing. Jan Černocký  
vedoucí ústavu

## Abstrakt

Tato bakalářská práce se zabývá tvorbou grafického intra s omezenou velikostí do 64kB. K vytvoření intra se využívá OpenGL API a jeho nadstavby. V práci jsou popsány techniky pro generování grafického obsahu, zahrnující procedurální generování, Perlinův šum, mipmapping, volumetrické paprsky, face culling, instancing, mlhu, skybox, a metody pro snížení velikosti výsledného spustitelného souboru. Výsledkem je grafické intro obsahující záběry z nočního města.

## Abstract

This Bachelor's thesis deals with the creation of a graphics intro with the executable file size limited to 64kB. The graphics intro is created using the OpenGL API and related libraries. The thesis describes many techniques for a graphics content generation including procedural generation, Perlin noise, mipmaps, volumetric light, face culling, instancing, fog, skybox, and techniques used for reduction of a size of the executable files. The result is a graphical intro showing a night city.

## Klíčová slova

grafické intro, 64kB, OpenGL, procedurální generování, Perlinův šum, mipmapping, volumetrické paprsky, face culling, instancing, mlha, skybox, Catmull-Rom křivka, exe packer, noční město

## Keywords

graphics intro, 64kB, OpenGL, procedural generation, Perlin noise, mipmaps, volumetric light, face culling, instancing, fog, skybox, Catmull-Rom spline, exe packer, night city

## Citace

MENŠÍK, Jakub. *Grafické intro 64kB s použitím OpenGL*. Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Michal Matýšek

# Grafické intro 64kB s použitím OpenGL

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Michala Matýška. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....

Jakub Menšík  
16. května 2018

## Poděkování

Rád bych poděkoval panu Ing. Michalu Matýškovi za odborné vedení a okamžitou pomoc kdykoliv bylo potřeba.

# Obsah

<b>1</b>	<b>Úvod</b>	<b>2</b>
<b>2</b>	<b>Použité techniky</b>	<b>3</b>
2.1	Procedurální generování . . . . .	3
2.2	Perlinův šum . . . . .	4
2.3	Mipmapping . . . . .	5
2.4	Volumetrické paprsky . . . . .	6
2.5	Face culling . . . . .	8
2.6	Instancing . . . . .	8
2.7	Mlha . . . . .	9
2.8	Skybox . . . . .	9
2.9	Catmull-Rom spline . . . . .	10
<b>3</b>	<b>OpenGL</b>	<b>13</b>
3.1	Shader . . . . .	13
3.2	Zobrazovací řetězec . . . . .	14
<b>4</b>	<b>Použité knihovny</b>	<b>16</b>
4.1	GLFW . . . . .	16
4.2	glad . . . . .	16
4.3	GLM . . . . .	16
4.4	libv2 . . . . .	16
<b>5</b>	<b>Implementace</b>	<b>17</b>
5.1	Generování textur . . . . .	17
5.2	Měsíční paprsky . . . . .	18
5.3	Budovy . . . . .	20
5.4	Pohyb kamery . . . . .	23
5.5	Hudba . . . . .	24
<b>6</b>	<b>Metody pro snížení velikosti aplikace</b>	<b>25</b>
6.1	Nastavení překladače . . . . .	25
6.2	Exe packer . . . . .	26
<b>7</b>	<b>Vyhodnocení</b>	<b>27</b>
<b>8</b>	<b>Závěr</b>	<b>30</b>
	<b>Literatura</b>	<b>31</b>

# Kapitola 1

## Úvod

Cílem této bakalářské práce je vytvořit grafické intro do 64kB s použitím OpenGL. Pod pojmem grafické intro, nebo také grafické demo, si můžeme představit neinteraktivní multi-mediální prezentaci. Jedná se o spustitelný soubor, který tvoří svůj obsah v reálném čase, čímž se liší například od animací, které mají svůj obsah předem dány. Existuje spousta soutěží, kde se programátoři snaží předvést svou kreativitu, své programátorské schopnosti a vytvořit co nejlepší grafické intro. Americký magazín *Wired News* často nazývá grafická demo jako „digitální graffiti“, *Digitalcraft* zase označil grafické demo jako „digitální origami“.

Tvorba grafických dem má často různá omezení, čímž vzniká několik soutěžních kategorií. Nejčastěji jsou to omezení týkající se velikosti spustitelného souboru, což je pro programátory mnohem větší výzva. Nejvíce soutěží probíhá v kategoriích do 1kB, 4kB nebo 64kB, mezi nejznámější patří například Assembly<sup>1</sup> nebo Revision<sup>2</sup>. Touto kategorií, grafické intro do 64kB, přesněji do 65536 bajtů, se zabývá i tato práce. Pro dosažení minimální velikosti je potřeba využít techniky jako například procedurální generování, syntézy zvuku nebo kompresi spustitelných souborů.

Procedurálnímu generování a dalším technikám pro tvorbu obsahu se věnuje kapitola 2, v kapitole 3 si popíšeme OpenGL, jeho výhody a postupy při vykreslování. V kapitole 4 si představíme použité knihovny, samotnou implementaci a využitím jednotlivých technik v této práci se zabývá kapitola 5 a v kapitole 6 si rozebereme hlavní možnosti, jak snížit výslednou velikost spustitelného souboru. Kapitola 7 obsahuje vyhodnocení celé práce.

---

<sup>1</sup>Více o soutěži Assembly Summer 2018 na stránkách <https://www.assembly.org/summer18>.

<sup>2</sup>Více o soutěži Revision a ukázky nejlepších grafických inter na jejich YouTube kanále <https://www.youtube.com/user/RevisionParty/videos>.

## Kapitola 2

# Použité techniky

Tato kapitola popisuje použité techniky v grafickém intru. Jedná se jak o techniky pro generování obsahu, tak o techniky vytvářející různé efekty jako mlha či volumetrické paprsky.

### 2.1 Procedurální generování

Procedurální generování je algoritmické vytváření obsahu s limitovaným nebo nepřímým uživatelským vstupem, jinými slovy je to počítačový software, který dokáže sám, nebo za pomoci jednoho nebo více uživatelů, vytvářet obsah [18]. Pod pojmem obsah si můžeme představit téměř vše, co je obsaženo v hrách, počítačových demech nebo animovaných filmech, jsou to textury, věci jako zbraně, zbroj, vozidla, dále hudba, úkoly, charaktery, mapy, úrovně a další [15].

Hlavní výhodou procedurálního generování je to, že není potřeba lidských designérů, kteří jsou drazí, a jejichž práce trvá dlouho. S postupem času je typické, že se na vývoji her nebo animovaných filmů podílejí stovky lidí, z nichž většina jsou designeři. Pokud se podaří napsat algoritmus, který dokáže většinu designerů nahradit, je to pro vývoj levnější a rychlejší. Další výhodou generování je to, že může vytvořit obsah, který nedokáží vymyslet ani kreativní designeři.

Tento způsob generování najde uplatnění především v herním průmyslu. Mezi nejznámější hry, které používají procedurální generování, patří No Man's Sky, která pomocí procedurálního generování vytváří celý vesmír, který obsahuje 18 trilionů planet, dále generuje veškerou faunu, flóru, kosmické lodě, atd, viz obrázek 2.1. Prozkoumání takového vesmíru by trvalo miliardu let. Hra Spore využívá procedurální generování k vytvoření nejrozmanitějších druhů živočichů, kteří se vyvíjejí od mikroskopických buněk, ale také generuje jejich pohyb a svět kolem nich. Série hry Diablo generuje podzemní chodby a různé odměny (zbraně, brnění, ...), které padají za zabití nepřátel. Další hry využívající procedurální generování jsou například Minecraft, Left 4 Dead, Borderlands, Civilization a další.

Existuje také řada vynikajících grafických dem, využívajících procedurální generování, které stojí za zmínku. Asi nejznámějším a nejpopulárnějším demem do 64kB je *Chaos Theory*<sup>1</sup> vytvořené maďarskou skupinou *Conspiracy*. Toto čtyřminutové demo s velmi dravou atmosférou a rázným průběhem obsadilo druhé místo v soutěži Assembly 2006, bylo nominováno na mnoho cen v různých kategoriích a dokonce bylo zmíněno i na filmových oceněních. Dalším velmi uznávaným demem je *fermi paradox*<sup>2</sup> od skupiny *Mercury*, které

<sup>1</sup>Demo je ke shlédnutí na <https://www.youtube.com/watch?v=ZfuierUvx1A>.

<sup>2</sup>Demo je ke shlédnutí na <https://www.youtube.com/watch?v=CnVtJ620jVU>.

obsadilo první místo na soutěži Revision 2016. Toto demo z kosmického prostředí zaujme diváky zajímavými planetami, ale také nesmírně detailním provedením povrchu planety, ze které je několik záběrů.



Obrázek 2.1: Ukázka ze hry No Man's Sky na jedné z vygenerovaných planet. Můžeme vidět vygenerované živočichy, rostliny a hory v pozadí.

## 2.2 Perlinův šum

V počítačové grafice lze pomocí geometrie dosáhnout skvěle vypadajících přesných tvarů a objektů. Pokud však chceme dosáhnout reality, samotná geometrie nám stačit nebude, jelikož v reálném životě věci naprosto přesné a geometricky dokonalé nejsou. Pokud si představíme například dvacet let starý dům, pro vytvoření jeho naprosto realistické podoby musíme zohlednit opadanou omítku, zašlé zdi, neumytá okna, začouzený komín a další.

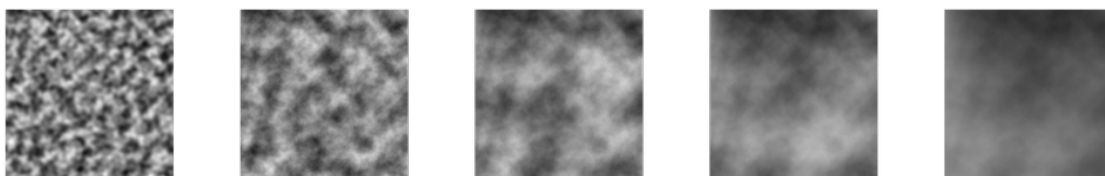
Na tento problém narazil i Ken Perlin, který se podílel na tvorbě filmu *Tron*. Všiml si „nedokonalosti“ dokonale renderovaných objektů ve filmu. Později, v roce 1985 uvedl ve své publikaci také metodu zvanou *šum*. Pod pojmem šum si většina lidí představí šum v televizi. Takový šum se však v grafice kvůli stále se náhodně měnícím hodnotám téměř využít nedá. Potřebujeme funkci, která generuje pro zvolený vstup pokaždé stejné hodnoty, které v sobě mají zahrnutý prvek náhodnosti. Zároveň musí jít o výpočetně nenáročnou funkci. Ken Perlin byl první, kdo s takovou funkcí přišel a je po něm pojmenovaná jako Perlinův šum. [16]

Oproti náhodnému šumu, který má náhodné, na sobě naprosto nezávislé hodnoty, Perlinův šum nabízí náhodnost hodnot, které jsou na sobě závislé. Zatímco přechod u normálního šumu je náhodný a skokový, u Perlinova šumu můžeme vidět jemný návazný přechod mezi body.



Perlinův šum má mnoho využití. Využívá se například při generování rozsáhlých procedurálních terénů při tvorbě her, dále najde využití při tvorbě procedurálních textur (ohně, vodní hladina, mraky, ...), při tvorbě materiálů (dřevo, kameny, ...), atd.

Principem Perlinova šumu je součet stejných šumových funkcí, které se ovšem liší ve svém měřítku a intenzitě. Tyto funkce se vytvoří spojitou interpolací hodnot z deterministického generátoru náhodných hodnot. Takto vytvořený šum má organický a přírodní vzhled. Počet součtů šumových funkcí je dán počtem oktáv, v každé oktávě má funkce různé měřítko a intenzitu. V praxi mají často funkce v každé oktávě dvojnásobnou frekvenci a poloviční amplitudu. Na obrázku 2.2 můžeme vidět výsledek Perlinova šumu při použití různého počtu oktáv.



Obrázek 2.2: Perlinův šum při použití různého počtu oktáv. Zleva je postupně použito 2, 4, 8, 16 a 32 oktáv.

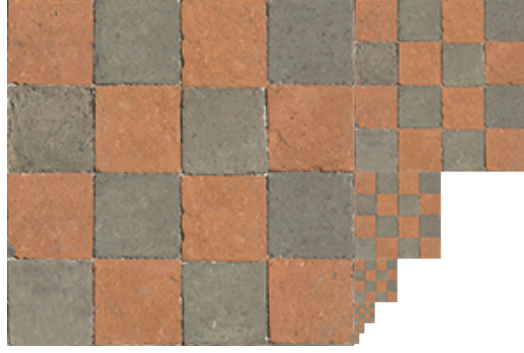
## 2.3 Mipmapping

Při pohybu těles s nanesenou texturou nebo při pohybu celé scény (při pohybu kamery) se mění vzdálenost objektů od pozorovatele a tím dochází, hlavně u vzdálených objektů, ke vzniku vizuálních chyb. Tyto chyby vznikají z toho důvodu, že se snažíme nanést velké textury na velmi malou plochu, čímž dochází ke špatnému výsledku vzorkování.

K minimalizování těchto vizuálních chyb se využívá technika zvaná mipmapping. Název mipmapping poprvé použil Lance Williams a vychází z prvních písmen latinského *Multrum In Parvo*, což znamená hodně věcí na malém místě a to je principem této techniky. Ten spočívá v tom, že si kromě hlavní textury ve velkém rozlišení uložíme taky verze s menším rozlišením. Jednotlivé verze s klesající velikostí jsou uloženy v hierarchické struktuře, viz obrázek 2.3. Velikosti textury je vhodné zmenšovat vždy na polovinu oproti naposledy uložené velikosti až po  $1 \times 1$  texel (rastrový element struktury). [16]

Při vykreslování je nejprve nutné zjistit relativní velikost povrchu polygonu vůči celé textuře a poté se vybere vhodné rozlišení textury, která se následně nanese. Další možností je vybrání nejbližší menší a nejbližší větší textury a barva pixelu se vypočte podle jednoho z následujících filtrů:

- *GL\_NEAREST\_MIPMAP\_NEAREST* - pro obarvení pixelu je vybrán nejbližší texel z nejbližší menší nebo nejbližší větší textury
- *GL\_NEAREST\_MIPMAP\_LINEAR* - provádí lineární interpolaci mezi nejbližšími texely z nejbližší menší a nejbližší větší textury
- *GL\_LINEAR\_MIPMAP\_NEAREST* - provádí bilineární interpolaci nejbližších texelů v jedné textuře



Obrázek 2.3: Hierarchické ukládání textur s polovičními velikostmi oproti naposledy uložené. Převzato z [7].

- *GL\_LINEAR\_MIPMAP\_LINEAR* - nejprve se provede interpolace pro výpočet barev texelů v obou texturách a poté se výsledná barva spočte další interpolací mezi dvěma předešlými.

Jednotlivé filtry mají různou rychlost a vizuální výsledky. Nejrychlejší je první možnost, zároveň však dosahuje nejhorších vizuálních výsledků. Opakem je poslední možnost, která je sice nejpomalejší, zato s nejlepšími výsledky. [6]

## 2.4 Volumetrické paprsky

Tato kapitola vychází ze článku [14]. Volumetrické paprsky, anglicky často nazývané jako God Rays, jsou 3D efekt, který se často uplatňuje ve scénách kvůli jednoduché implementaci a skvělému výsledku. V reálném životě si pod pojmem volumetrické paprsky můžeme představit slunce svítící skrz mraky, viz obrázek 2.4.

Abychom vypočítali osvětlení v nějakém bodě, musíme brát v úvahu rozptyl od světelného zdroje k danému bodu a zjistit, zda je bod za překážkou nebo ne. Začneme analytickým modelem pro rozptyl světla od Hoffmana a Preethama [13]. Mějme vzorec

$$L(s, \theta) = L_0 e^{-\beta_{ex} S} + \frac{1}{\beta_{ex}} E_{sun} \beta_{sc}(\theta) (1 - e^{-\beta_{ex} S}), \quad (2.1)$$

kde  $s$  je vzdálenost, kterou světlo prošlo a  $\theta$  je úhel který svírá paprsek se sluncem.  $E_{sun}$  je zdrojová světelnost slunce,  $\beta_{ex}$  je umírňovací konstanta skládající se z absorpce a rozptylování světla do prostoru.  $\beta_{sc}$  je úhlové rozptýlení. Důležitým aspektem této rovnice je, že její první část počítá, kolik světla bylo pohlceno mezi zdrojem světla a pozorovatelem a druhá část přičítá světlo, které bylo do této cesty rozptýleno. Efekt průchodnosti různými prostředími (mraky, budovami) je modelován pomocí snížení jasu po průchodu těmito objekty.

$$L(s, \theta, \phi) = (1 - D(\phi)) L(s, \theta), \quad (2.2)$$

kde  $D()$  je procento zakrytí.



Obrázek 2.4: Volumetrické paprsky v reálném životě. Převzato z [5].

Tato myšlenka představuje problém zjišťování absorpce světla ze zdroje do každého bodu obrázku. Na obrazovce v době vykreslování nemáme informace o celém prostoru vykreslované scény, abychom byli schopni přesně určit absorpci světla. Nicméně můžeme uvažovat, že jas bude odpovídat poměru vyslaného a absorbovaného směrem z každého bodu do zdroje světla. Poměr vzorků, které narazí na vyzařující oblast, proti vzorkům, které narazí na překážky, nám dá  $D()$ . Tato technika funguje velmi dobře v případech, kde je zdroj světla velmi jasný v porovnání s ostatními objekty.

Pokud podělíme osvětlení vzorku počtem vzorků  $n$ , *post-process* zpracování nám dá aditivní vzorkování obrazu:

$$L(s, \theta, \phi) = \sum_{i=0}^n \frac{L(s_i, \theta_i)}{n}, \quad (2.3)$$

Nakonec přidáme do rovnice koeficienty, abychom parametrizovali kontrolu součtu. Výsledná rovnice vypadá takto:

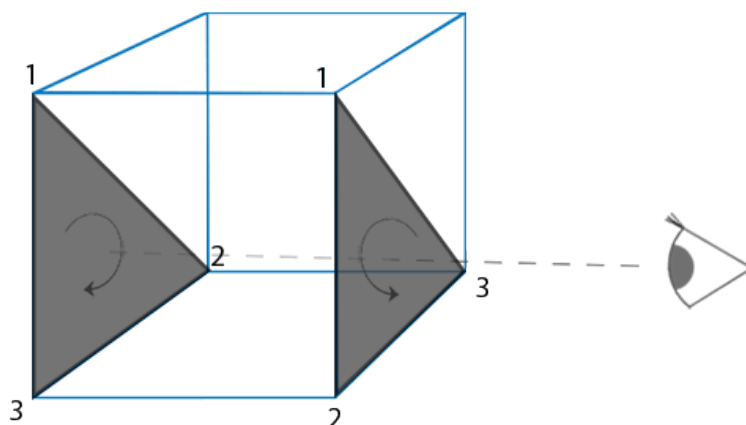
$$L(s, \theta, \phi) = exposure \times \sum_{i=0}^n decay^i \times weight \times \frac{L(s_i, \theta_i)}{n}, \quad (2.4)$$

kde *exposure* řídí celkovou intenzitu post-processu, *weight* řídí váhu každého vzorku a  $decay^i$  rozptýlí příspěvek každého vzorku s tím, jak se paprsek posouvá dále od světelného zdroje. Jednoduše řečeno, *exposure* a *weight* řídí výsledný jas. Tuto metodu je možné použít i s více světelnými zdroji.

## 2.5 Face culling

Face culling je technika, která se snaží co nejvíce omezit počet fragmentů, které vykreslujeme na obrazovku. Pokud si představíme například kvádr, vidíme maximálně pouze tři strany z osmi a je tedy zbytečné všech osm stran vykreslovat, když nám stačí vykreslit tři.

Principem této techniky je zjistit, zda je vykreslovaná část natočená ke kameře, nebo ne. Pokud není natočená ke kameře, nedojde k její rasterizaci, čímž se ušetří práce pro fragment shader. Jakým směrem je otočená zjistíme podle definice vrcholů, zda jsou definovány po směru hodinových ručiček, nebo proti. Při vytváření objektů je tedy potřeba určit, jak budou vrcholy definovány. GPU pak při vykreslování kontroluje, zda vrcholy odpovídají definici, nebo jsou v opačném směru, viz obrázek 2.5.



Obrázek 2.5: Princip fungování face cullingu. Na okem neviditelné straně krychle jsou vrcholy definovány v opačném směru než na okem viditelné straně. Převzato z [7].

## 2.6 Instancing

Instancing je metoda, která slouží k vykreslování téměř stejných objektů několikrát po sobě. Kdybychom chtěli vykreslit například obrovské parkoviště nově vyrobených aut, které obsahuje tisíce aut stejného typu lišících se pouze v barvě a pozici na parkovišti, vykreslení každého auta zvlášť by bylo velmi neefektivní a časově náročné. Místo toho můžeme využít instancing, který vykreslí všechna auta jedním příkazem a minimalizuje tak časovou náročnost vykreslení.

Tento příkaz pro vykreslení vypadá podobně jako příkazy bez použití instancingu. Nejprve si určíme vertexy, podle kterých budeme vykreslovat. Poté přidáme k vertexům atributy. Některé atributy se mezi konkrétními instancemi měnit nebudou, například pozice vertexu, jiné se měnit budou, například barva, transformační matice, atd. Pro takové atributy si vytvoříme buffer, do kterého uložíme všechny hodnoty a poté určíme, jak často se mají tyto atributy při instancingu měnit. Pro příklad si představme, že máme typ auta, který chceme rozmístit na tisíc míst na parkovišti tak, že bude mít pět různých barev. Vytvoříme si tedy dva pomocné buffery, jeden pro barvu, kam si uložíme pět různých barev, a druhý pro transformační matice, kam si uložíme tisíc různých matic, které nám rozmístí

auta různě po parkovišti. Pak už stačí nastavit, aby se barva vyměnila po vykreslení dvou set instancí a aby se transformační matice změnila po každém vykreslení instance.

## 2.7 Mlha

Použití mlhy je velmi populární technikou využívanou v počítačové grafice. Hlavním úkolem mlhy je dát divákovi informaci o tom, v jaké vzdálenosti se nachází jednotlivé objekty scény. Kromě toho nám může mlha o prostředí říct mnohem více. Změnou tradiční namodralé barvy na nažloutlou barvu, když je náš pohled srovnáný se směrem svitu slunce, můžeme přidat větší realismus. Se správným použitím mlhy jde také napodobit záři a další světelné efekty bez použití víceprůchodového vykreslování.

K vytvoření mlhy si musíme určit barvu mlhy a poté spočítat, jak velký poměr mlhy chceme v daném bodě použít. Existuje několik vzorců, jak poměr mlhy v bodu spočítat. Lineárně lze poměr spočítat funkcí

$$fogAmount = distance * b, \quad (2.5)$$

kde  $fogAmount$  je poměr mlhy,  $distance$  je vzdálenost bodu od kamery a  $b$  je námi vhodně zvolený koeficient určující hustotu mlhy. Tato funkce je výpočetně nejjednodušší a poměr mlhy lineárně stoupá se vzdáleností od kamery. Exponenciálně lze poměr mlhy spočítat vzorcem

$$fogAmount = 1 - e^{(-distance*b)}. \quad (2.6)$$

Tento poměr mlhy má exponenciální nárůst, jeho použití můžeme vidět na obrázku 2.7. Další funkce využívá exponenciální nárůst s exponentem umocněným na druhou. Oproti exponenciálnímu nárůstu je nárůst prudší.

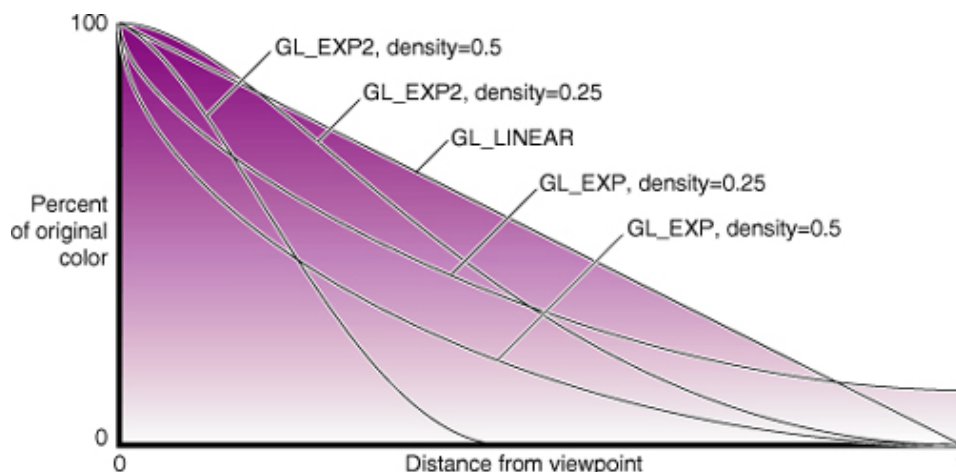
$$fogAmount = 1 - e^{-(distance*b)^2}. \quad (2.7)$$

Porovnání těchto funkcí lze vidět na obrázku 2.6. Výslednou barvu pixelu spočítáme lineární interpolací mezi barvou pixelu a barvou mlhy podle vzorce

$$color = originalColor * (1 - fogAmount) + fogColor * fogAmount. \quad (2.8)$$

## 2.8 Skybox

Skybox je technika, která vytváří pozadí scény a dává scéně mnohem lepší dojem. Jedná se o krychli, která se skládá ze šesti samostatných textur, které zobrazují okolí a dávají divákovi dojem, že se nachází v obrovském prostředí, i když tomu tak ve skutečnosti není.



Obrázek 2.6: Porovnání mezi jednotlivými funkcemi pro výpočet poměru mlhy. Převzato z [17].

Typicky se na strany skyboxu promítají hory, obloha, země, moře atd. Na obrázku 2.8 můžeme vidět ukázkou skyboxu.

U skyboxu je velmi důležitá návaznost jednotlivých stran krychle, aby nebylo poznat, že se jedná o krychli, ale aby skybox dával dojem skutečného okolí. Tento dojem můžeme podpořit deformací bodů, které se nacházejí blízko vrcholů a hran.

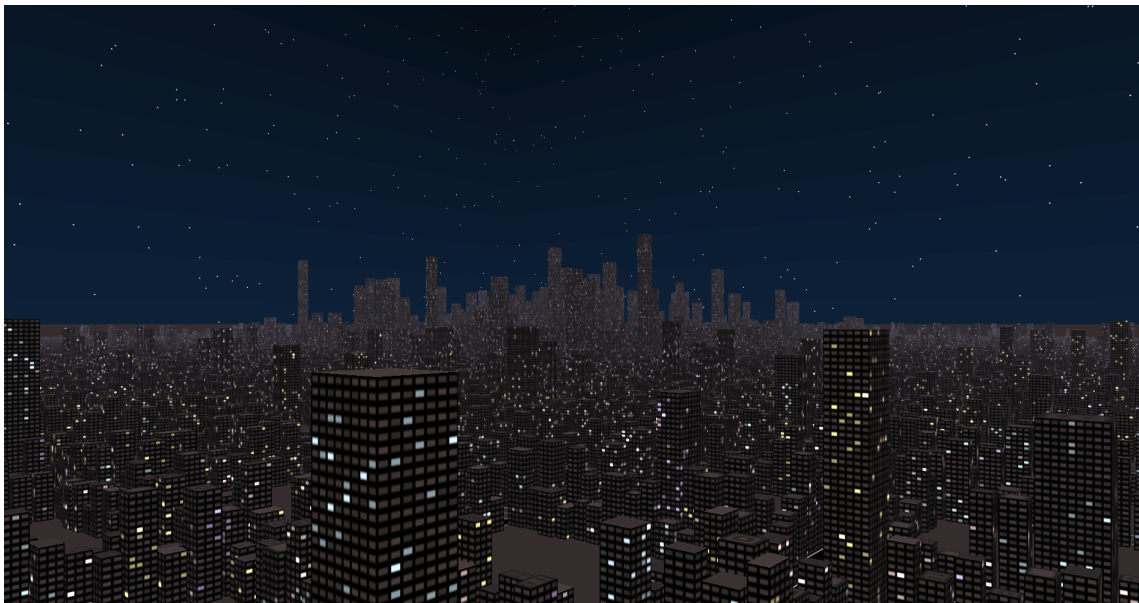
Skybox má i několik nevýhod, jednou z nich je to, že se všechny objekty zdají být nekonečně daleko, i když tomu tak nemusí být. V dnešní době se občas využívají i tzv. skydome, což jsou koule či polokoule, na které se nanášejí textury.

## 2.9 Catmull-Rom spline

Křivky se v grafice využívají k popisu „křivých tvarů“, jelikož ideální vlastnosti jako kulatost kružnic nebo rovnost přímek se ve skutečnosti příliš nevyskytují. Takové tvary mají téměř všechny objekty, se kterými přicházíme v reálném životě do styku. V grafice využíváme křivky k definici objektů, fontů, k určování dráhy objektů, ale také k určování trajektorie kamery, což je pro grafické intro velmi podstatné.

Křivky mají spoustu vlastností, kterými se liší. Pro pohyb kamery je pro nás důležité hlavně to, aby byla křivka interpolační. Interpolační křivky, na rozdíl od aproximačních, procházejí body. Dále je pro nás důležitý hladký přechod mezi jednotlivými body, proto je vhodné zvolit spline křivku, která tento hladký přechod zaručí. Jako nejvhodnější křivka pro určení trajektorie kamery je na základě požadovaných vlastností Catmull-Rom spline.

Catmull-Rom spline je interpolační křivka, kterou formuloval Edwin Catmull a Raphael Rom. Je definovaná  $N$  body, přičemž křivka vychází z druhého bodu a končí v bodě předposledním, viz obrázek 2.9. Aby křivka procházela všemi body, je potřeba krajní body zdvojit. Hladký přechod mezi jednotlivými segmenty zaručuje to, že tečný vektor v bodě  $P_i$  je rovnoběžný s vektorem  $P_{i-1} - P_{i+1}$ . Catmull-Rom křivku můžeme zapsat maticově jako



Obrázek 2.7: Použití mlhy s exponenciálním nárůstem.

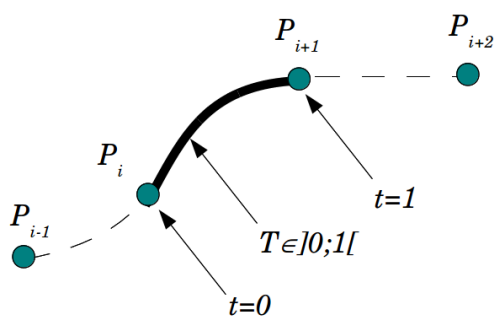
$$P(t) = \frac{1}{2}[t^3, t^2, t^1, 1] \begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_{i-3} \\ P_{i-2} \\ P_{i-1} \\ P_{i-0} \end{bmatrix}, \quad (2.9)$$

kde  $t$  je hodnota v intervalu  $< 0, 1 >$ , která udává, v jaké části mezi dvěma body se zrovna nacházíme.





Obrázek 2.8: Skybox použitý v tomto grafickém intru. Jelikož je intro zasazeno do noční doby, je skybox tmavý, na obloze jsou vidět hvězdy a velký měsíc. Spodní půlka skyboxu znázorňuje zem.



Obrázek 2.9: Ukázka Catmull-Rom křivky. Převzato z [1].



## Kapitola 3

# OpenGL

OpenGL (z anglického Open Graphics Library) je API vyvinuta firmou Silicon Graphics Inc. v letech 1991 až 1992. Jedná se o multiplatformní standard pro tvorbu převážně 3D grafických aplikací. Nejnovější verze OpenGL 4.6 byla vydána 31.6.2017 a jedná se již o osmnáctou verzi této grafické knihovny. Pod pojmem API si můžeme představit specifikaci hromady funkcí, které voláme. Nejedná se tedy přímo o kód, API nám pouze říká, jaké funkce existují, jaké přijímají parametry a co vracejí. V tomto případě, jelikož se jedná o grafické API, nám OpenGL umožňuje volat grafické funkce a pracovat přímo s grafickou kartou (GPU). O implementaci těchto grafických funkcí se stará přímo výrobce grafických karet (NVIDIA, AMD, Intel, ...), což znamená, že na každé grafické kartě může být trochu jiná implementace některých funkcí a tedy i trochu jiný výsledek vykreslování.

Kromě OpenGL existuje řada dalších grafických API umožňujících práci s grafickou kartou, jako například Direct3D, Metal, Vulkan, Mantle a další. Výhodou OpenGL je možnost použití na různých typech grafických karet, dokonce lze použít bez nainstalované grafické karty za použití softwarové simulace. Další výhodou je multiplatformní využití na vícero systémech, které nemůžou nabídnout například Direct3D nebo Metal, které jsou zaměřeny pouze na Windows, respektive Mac. To je výhoda zejména pro začátečníky a menší projekty, které si nemůžou dovolit velký tým pracujících s různými API pro jednotlivé systémy. Obecně se ale ve větších herních společnostech vytváří projekty s různými API (Direct3D pro Windows, Metal pro Mac, ...) pro dosažení lepších výsledků v daném systému. Na rozdíl od jiných grafických sad OpenGL neobsahuje funkce pro práci s okny nebo funkce pro zpracování událostí. Pro tyto potřeby se využívá dalších knihoven, které slouží jako nadstavba OpenGL, viz kapitola 4.1.

OpenGL od svého vzniku prošlo řadou změn, a proto se často setkáme s označením staré OpenGL a moderní OpenGL. Staré OpenGL nabízelo několik předvolených funkcí, které vývojářům neumožňovaly velkou kontrolu nad výsledkem, například pro přidání stínování se jednoduše povolilo stínování a o zbytek už se postaraly funkce, nikoli vývojář. Moderní OpenGL na druhou stranu nabízí vývojářům velkou kontrolu a možnosti. Hlavní změnou oproti staré verzi je přidání tzv. *shaderů*.

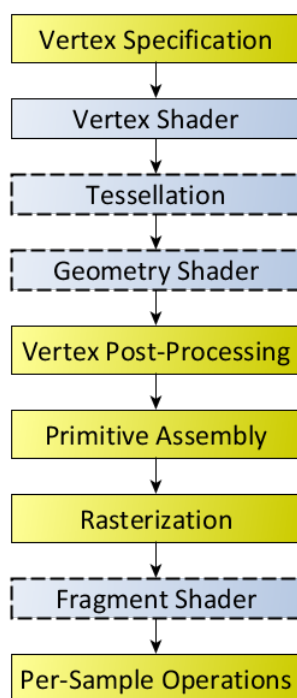
### 3.1 Shader

Shader je program, který neběží na CPU, jako programy napsané v C++, Javě, atd., ale běží na GPU. Je to z toho důvodu, že hodně výpočtů probíhá na GPU rychleji a zároveň chceme naši grafické kartě říct, jak se má chovat například při stínování, osvětlení atd. Mezi

základní a nejčastěji používané shadery patří vertex shader (shader vrcholů) a fragment shader (shader fragmentů), ale existují také méně používané shadery jako geometry shader, tessellation shader nebo compute shader, které se hodí při použití pokročilejších technik.

## 3.2 Zobrazovací řetězec

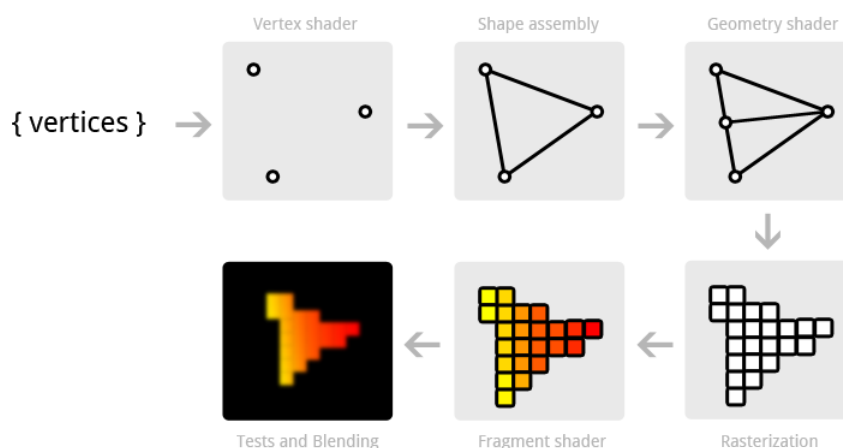
Pro reprezentaci dat se v OpenGL používají tzv. *vertexy* neboli vrcholy. Každý vertex může obsahovat několik atributů, nejčastěji se jedná o pozici, texturovací koordináty, barvu a další. Z těchto vertexů se poté skládají další primitiva jako úsečky, trojúhelníky, atd. Tyto vertexy jsou uloženy jako seznam ve VBO (vertex buffer object).



Obrázek 3.1: Jednotlivé kroky při vykreslování objektů v OpenGL. Položky se žlutým pozadím nejsou programovatelné. Převzato z [11].

Zobrazovací řetězec představuje abstrakci způsobu zpracování dat na grafické kartě. Skládá se z několika programovatelných a neprogramovatelných úrovní. Programovatelné úrovně se označují pojmem shadery. Do paměti si ukládáme vertexy, které jsou později zpracovávány při průchodu skrze vykreslovací řetězec. Dále si zde můžeme uložit textury, framebuffer (rámce dat s hodnotami barev pro jednotlivé pixely na obrazovce) nebo například transformační matice. Dříve obsahovaly grafické karty dva typy procesorů, a to zvlášť pro vertex shader a zvlášť pro fragment shader, což způsobovalo menší flexibilitu systémů. V dnešní době se však výpočetní část provádí na procesorech, které jsou unifikované, to znamená, že je to jeden typ výpočetní jednotky, na který se mapuje provádění jednotlivých shaderů. Toto řešení je mnohem flexibilnější, můžeme si alokovat výpočetní jednotky podle toho, jaký druh shaderu potřebujeme v danou chvíli více využívat. Na začátku je spuštěn vertex shader. Ten má na vstupu jednotlivé vertexy, se kterými provádí různé vý-

počty, typicky transformace a násobení matic. Vertexy se zpracovávají paralelně, nejsme tedy schopni ovlivnit pořadí jejich zpracování. Po zpracování tří vertexů dojde k sestavení základního primitiva, trojúhelníku, a k ořezání v *clip space*. Následně dochází k rasterizaci, kde se požadované atributy vrcholů rozinterpolují po trojúhelníku do jednotlivých pixelů. Následuje fragment shader, který provádí různé výpočty nad jednotlivými fragmenty, například výpočet osvětlení, a dochází k obarvení fragmentů. Poslední důležitou fází před vykreslením jsou per fragment operace, což jsou operace pro každý fragment. Asi nejdůležitější operací je testování hloubky, kdy chceme zjistit, který objekt je na daném pixelu v popředí. Každý pixel má uloženou hloubku nejbližšího objektu, který byl na daném pixelu zpracován a jakmile najdeme fragment, který má menší hloubku, než je uložená hloubka v daném pixelu, aktualizujeme v daném pixelu barvu i hloubku. Dojde tak k zapsání fragmentu do framebufferu. Jak již bylo zmíněno výše, existují i jiné shadery, pro nás jsou však důležité hlavně vertex a fragment shader. Vertex shader je spouštěn tolikrát, kolik máme vertexů. Fragment shader je většinou spouštěn mnohem častěji, jelikož se spouští pro každý fragment, proto je důležité provádět všechny výpočetní operace, které nepotřebují výpočet pro každý fragment, ve vertex shaderu, čímž maximalizujeme výkon. Celý zobrazovací řetězec můžeme vidět na obrázku 3.1, zpracování vertexů až po vykreslení trojúhelníků můžeme vidět na obrázku 3.2.



Obrázek 3.2: Ukázka vykreslení trojúhelníku od vrcholů až po vykreslené objekty na obrazovce. Převzato z [9].

## Kapitola 4

# Použité knihovny

Při implementaci si se samotným OpenGL nevystačíme a je tedy nutné použít různé knihovny. Tato kapitola obsahuje informace o knihovnách použitých v grafickém intru.

### 4.1 GLFW

GLFW [4] je knihovna napsaná v jazyce C. Je zaměřená speciálně na OpenGL, poskytuje základní funkce pro renderování objektů na obrazovku a vytváření OpenGL obsahu. Dále slouží pro práci s okny, podporuje více monitorů, high-DPI monitory, dále nám umožňuje zpracovávat uživatelské vstupy jako myš, klávesnice, gamepad, zpracování událostí, zpětných volání a další.

### 4.2 glad

Knihovna glad [3] slouží k načtení funkcí OpenGL za běhu aplikace, jelikož lokace většiny funkcí není v době překladu známá. Glad tedy ukládá ukazatele na tyto funkce pro pozdější využití. Často se také využívá známější knihovna GLEW.

### 4.3 GLM

Knihovna GLM [10] (z anglického OpenGL Mathematics) je matematická knihovna v jazyce C++, která je určená pro grafické programy využívající OpenGL a GLSL (z anglického OpenGL Shading Language), což je vyšší programovací jazyk pro psaní shaderů. Knihovna nabízí nespočet užitečných funkcí, například umožňuje počítat maticové transformace, kvaterniony, dále nabízí generování náhodných čísel či šumu.

### 4.4 libv2

Knihovna libv2 [2] je vyvinuta skupinou Farbrausch speciálně pro grafické intra s omezenou velikostí. Obsahuje syntetizátor zvuku, který umožňuje přehrávání \*.v2m souborů, což jsou audio soubory rovněž vyvinuty skupinou Farbrausch. Tento typ souborů má uloženou sérii instrukcí, kterou má za úkol syntetizátor rozkódovat a přehrát jako hudbu. Jelikož je část knihovny napsaná v jazyce assembler, je nutné přidat do řešení překladač pro assembler. Takových překladačů je několik, v tomto intru byl použit YASM, který je možné stáhnout z [12].

## Kapitola 5

# Implementace

Tato kapitola popisuje implementační detaily grafického intra. Je zde popsáno generování textur, generování budov a dalšího obsahu, rozložení města, implementace pohybu kamery a vložení hudby do intra.

### 5.1 Generování textur

Texturování slouží k dodání reálnějšího vzhledu objektům ve scéně. Jelikož je cílem vygenerovat celý obsah grafického intra, nejsou použity žádné externí textury, všechny jsou vygenerovány po spuštění programu. Tato sekce popisuje generování textur budov a generování skyboxu s měsícem. V intru jsou také generovány textury silnic, ale ty nejsou v této sekci kvůli jednoduchosti popsány. Nanášení textur na objekty má na starost fragment shader, který jsme si popsali v kapitole 3.2.

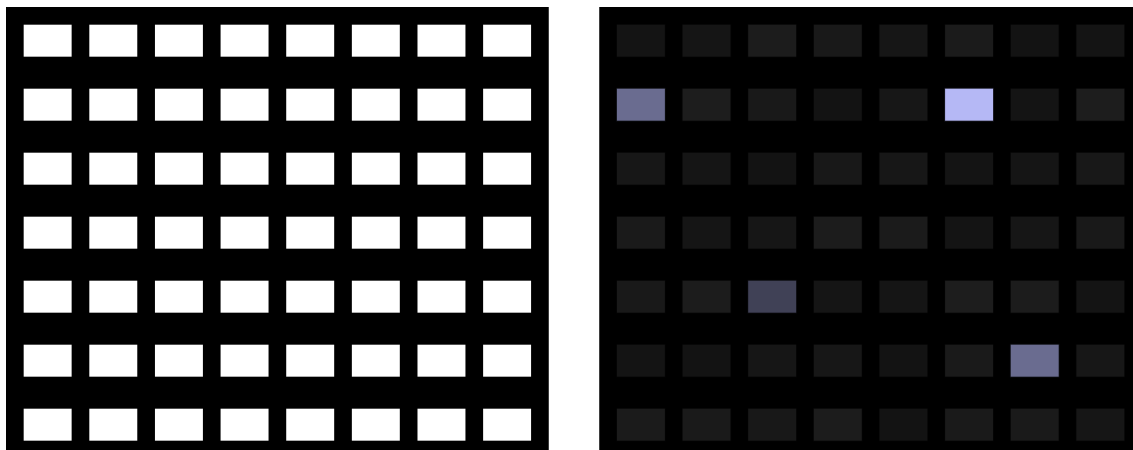
#### 5.1.1 Textury budov

Při texturování budov je potřeba využít dvou různých textur, jednu pro strany domu a druhou pro střechu. Střešních textur je vygenerováno pouze šest, protože není nutnost velkých rozdílů. Jedná se o šedé plochy doplněné o různé tmavší pruhy. Vytváření bočních textur budov je inspirováno grafickým intrem *Pixel City*<sup>1</sup>. Základní textury s bílými okny můžeme vidět na obrázku 5.1. Při spuštění programu se vytvoří několik desítek textur, které se liší výplní bílých oken. Okna jsou náhodně vybarvována buďto tmavou barvou nebo světlejší šedou až bílou barvou. O přidání fialové či jiné barvy do světlých či bílých míst se stará až fragment shader podle toho, jakou barvu má vykreslovaná budova přiřazenou. Na všechny textury je použita metoda mipmappingu, viz kapitola 2.3, která slouží k odstranění aliasingu. Pro vygenerování mipmapových textur nám stačí zavolat OpenGL funkci `glGenerateMipmaps`, která vytvoří všechny zmenšené textury za nás a tím ušetří množství kódu.

Generování textur budov, ale také silnic, má na starost třída `Texture`, která je vytvořena podle návrhového vzoru *Singleton*. Tento návrhový vzor se používá v případech, kdy potřebujeme, aby v celém programu byla vytvořena pouze jedna instance třídy, typicky se využívá při uložení nastavení ve hrách (hlasitost zvuku, rozlišení, jas, ...). Po spuštění programu se tato instance vytvoří a zároveň se vygenerují všechny textury. Při vykreslování můžeme jednoduše z jiných tříd tuto instanci požádat o přiřazení textury.

---

<sup>1</sup>Grafické intro Pixel City je k prohlédnutí na <https://www.youtube.com/watch?v=-d2-PtK4F6Y>.



Obrázek 5.1: Na obrázku můžeme vidět malou část základní textury budov (vlevo) a náhodně vybarvenou texturu, která může mít různé barvy a různě rozsvícená okna (vpravo).

### 5.1.2 Skybox a měsíc

Textury pro skybox jsou generovány v třídě **Skybox**. Je potřeba vygenerovat šest stran krychle, které tvoří stěny skyboxu. Spodní strana je celá v hnědé barvě a znázorňuje zem. Boční strany jsou do půlky také v hnědé barvě země. Od půlky výše následuje barevný přechod od tmavě modré barvy do černé. Vrchní strana je černá. Všude kromě země se náhodně přidají bílé tečky znázorňující hvězdy.

Tím generování skyboxu nekončí, ještě je potřeba vygenerovat měsíc. Měsíc obsahuje na svém povrchu světlejší i tmavší skvrny, které se nazývají měsíční moře a jsou to prohlubně po srážkách s planetkami, které byly zality lávou, která ztuhla. Pro vytvoření těchto skvrn se nám hodí použití šumu. Aby měly skvrny organický a reálnější tvar, využijeme Perlinův šum, viz kapitola 2.2. Nakonec kolem měsíce přidáme lehkou záři. Výsledný měsíc je možné vidět na obrázku 5.2, celý skybox pak na obrázku 2.8.

Pro výpočet volumetrických paprsků je potřeba vygenerovat ještě jeden skybox, kde je měsíc jako zdroj světla vybarven bílou barvou a všechno ostatní je černé. Implementace těchto paprsků je popsána v kapitole 5.2.

## 5.2 Měsíční paprsky

Jedním z nejzajímavějších efektů v tomto grafickém intru jsou volumetrické paprsky, které znázorňují svit měsíce zpoza budov. Výpočet těchto volumetrických paprsků je popsán v kapitole 5.2. V algoritmu 1 můžeme vidět výpočty, které provádí fragment shader, a které odpovídají vzorci 2.4. Z vertex shaderu si předáme pozici světla na obrazovce a texturové koordináty. Zde využijeme texturu s vykreslenou scénou a texturu, která obsahuje tzv. *occlusion map* (zdroj světla je vykreslen bíle, všechno ostatní černě). Následně proběhne vzorkování podle již zmíněného vzorce, počet vzorků si zvolíme tak, aby nebyla záře moc velká ani moc malá. Nakonec přičteme k barvě na daném fragmentu jeho původní barvu.



Obrázek 5.2: Textura měsíce zakomponovaná do skyboxu. Měsíc má na sobě jak světlé, tak tmavé skvrny, které znázorňují měsíční moře. Kolem měsíce je lehká záře.

---

#### Algoritmus 1 Volumetrické paprsky

---

```

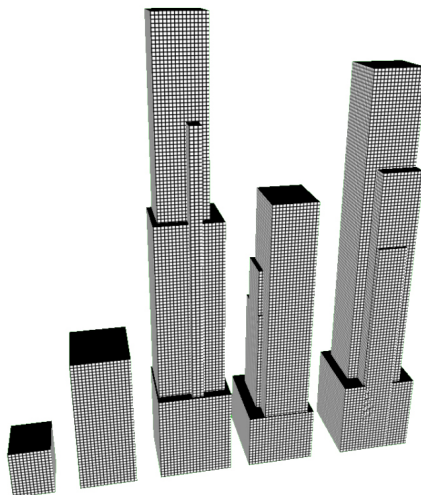
1: in vec2 TexCoords;
2: in vec2 LightPositionOnScreen;
3: out vec4 FragColor;
4:
5: uniform sampler2D occlusionMap;
6: uniform sampler2D renderedScreen;
7:
8: vec2 tc = TexCoords;
9: vec2 deltaTexCoord = tc - LightPositionOnScreen.xy;
10: deltaTexCoord *= 1.0 / float(NUM_OF_SAMPLES) * density;
11:
12: for (i=0; i<NUM_OF_SAMPLES; i++)
13: {
14:   tc -= deltaTexCoord;
15:   vec4 sample = texture2D(occlusionMap, tc) * 0.4;
16:   sample *= illuminationDecay * weight;
17:   color += sample;
18:   illuminationDecay *= decay;
19: }
20:
21: vec4 realColor = texture2D(renderedScreen, TexCoords);
22: FragColor = ((vec4((vec3(color) * exposure), 1)) + (realColor*(1.1)));

```

---

## 5.3 Budovy

V intru se nacházejí tři typy budov, které se liší svou velikostí i strukturou. Jedná se o domy, panelové domy a mrakodrapy, viz obrázek 5.3. Všechny budovy jsou uskupeny do bloků. Tato sekce se zabývá generováním těchto budov a problémy, které nastaly.



Obrázek 5.3: Různé typy budov v intru se základní texturou. Zleva dům, panelový dům a nakonec tři různé mrakodrapy.

### 5.3.1 Domy a panelové domy

Domy a panelové domy se v intru liší pouze velikostí, jejich struktura je stejná. Jedná se o kvádry, které se skládají ze dvou tzv. meshů, což jsou menší části objektu, které se vykreslují zvlášť. První mesh je plášť kvádra, který je tvořen deseti vertexy a dvaceti čtyřmi indexy. Plášť kvádra by šlo popsat pouze osmi vrcholy, my však potřebujeme dvojici vrcholů použít dvakrát, jednou jako počáteční koordináty textury a jednou jako koncové. Druhý mesh je horní podstava kvádra, neboli střecha domu. Ta je tvořena čtyřmi vertexy a šesti indexy.

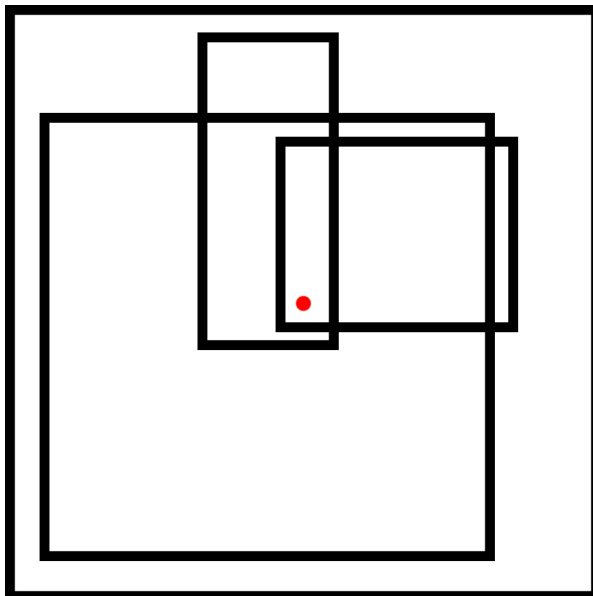
Původně byly všechny budovy v intru unikátní, ale to bylo velmi náročné na paměť, jelikož se musely ukládat informace o každé budově a těch je v intru téměř padesát tisíc. Proto se přešlo k řešení, kdy se na začátku vygeneruje několik set různých domů a panelových domů a ty se v intru opakují. Při tomto řešení se nabízí použití instancingu, viz kapitola 2.6. Pro každý dům se tedy vytvoří objekt, který obsahuje pole transformačních matic, podle kterých do města rozmístíme jednotlivé instance daného domu.

### 5.3.2 Mrakodrapy

Mrakodrapy, na rozdíl od domů a panelových domů, se liší jak velikostí, tak strukturou, a tedy každý mrakodrap v intru je unikátní. Mrakodrapy se skládají z několika menších částí, které mají stejnou strukturu jako domy (dva meshy). Princip generování mrakodrapů lze vidět na obrázku 5.4. Nejprve nalezneme střed podstavy mrakodrapu, následně si určíme



náhodný počet menších částí, kterých může být maximálně pět (maximálně může mít mrakodrap deset meshů). Poté generujeme menší části mrakodrapu tak, aby podstava vždycky obsahovala střed podstavy mrakodrapu. Tento způsob generování může vytvořit vizuální chybu zvanou z-fighting, viz kapitola 5.3.4.

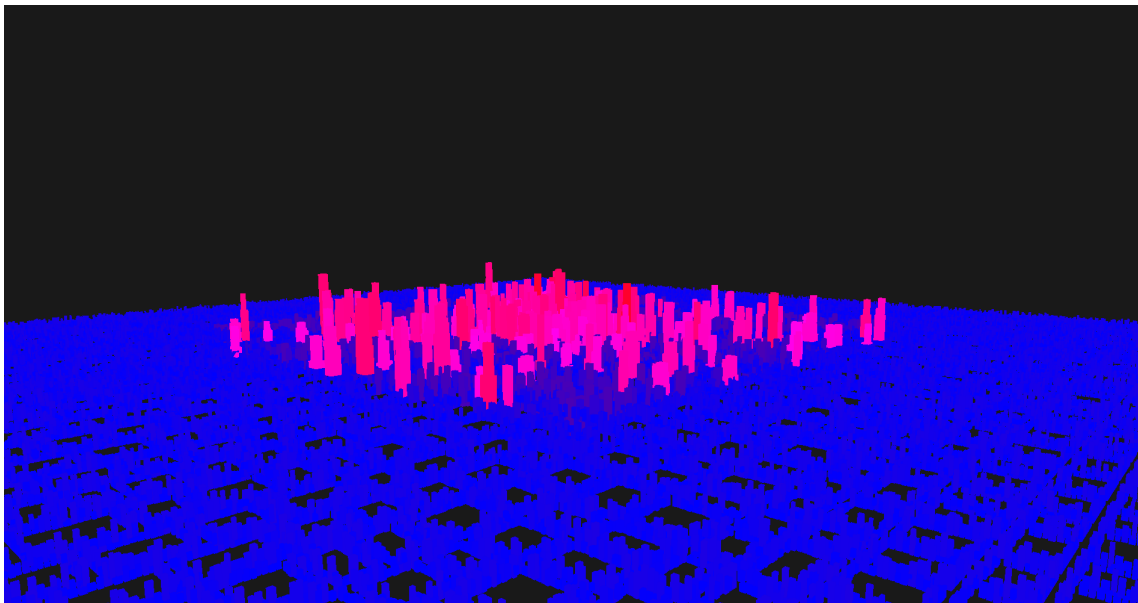


Obrázek 5.4: Podstava mrakodrapu. Černé obdélníky znázorňují podstavy menších kvádrů, ze kterých se skládá výsledný mrakodrap. Červenou tečkou je vyznačen střed, který musí náležet každé podstavě menších kvádrů.

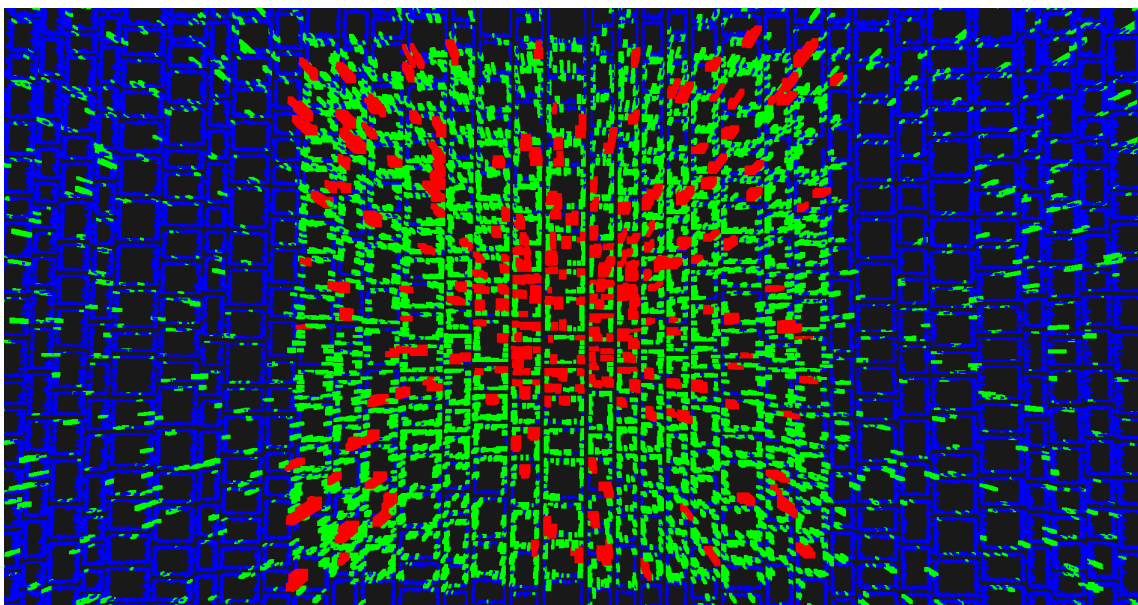
### 5.3.3 Rozložení města

Město je rozděleno do několika částí tak, aby co nejvíce odpovídalo rozložení skutečných měst. Na výškové mapě města Vancouver<sup>2</sup>, můžeme vidět, že nejvyšší budovy, mrakodrapy, jsou uskupeny u sebe v centru, a dále od centra jsou budovy postupně menší a menší. Stejně tak je to i v tomto intru, jak můžeme vidět na výškové mapě, viz obrázek 5.5. V každé části mají různé typy budov různou pravděpodobnost výskytu, v centru mají nejvyšší pravděpodobnost výskytu mrakodrapy, naopak na kraji města mají nulovou pravděpodobnost. Rozložení města můžeme vidět na obrázku 5.6.

<sup>2</sup>Výšková mapa města Vancouver k prohlédnutí na <http://maps.nicholsonroad.com/heights/>.



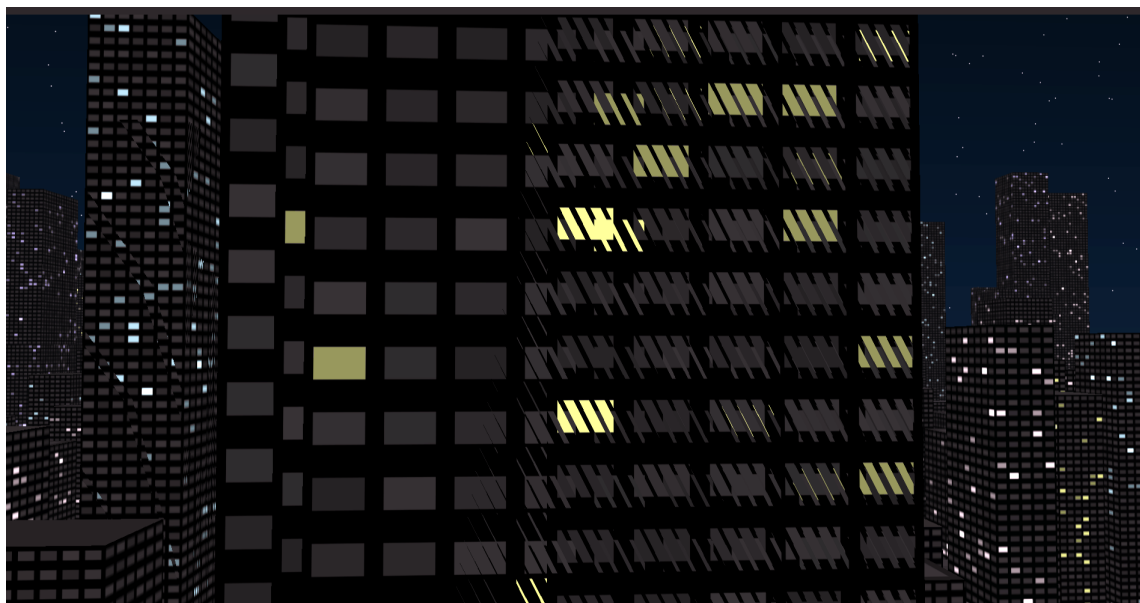
Obrázek 5.5: Výšková mapa výsledného města, kde červeně jsou vybarveny nejvyšší budovy a modře ty nejmenší. Nejvyšší budovy jsou umístěny v centru města, směrem dál od centra výška budov klesá.



Obrázek 5.6: Rozložení budov výsledného města, kde mrakodrapy jsou vybarveny červeně, panelové domy zeleně a malé domy modře. Město je rozděleno do čtyř oblastí, v každé mají různé typy budov různou pravděpodobnost výskytu.

### 5.3.4 Z-fighting

Z-fighting je jev, který se objevuje v 3D renderování. Nastává v momentě, kdy mají dva či více primitiv stejné nebo velmi blízké hodnoty v z-bufferu. V tomto intru tento problém nastává hlavně u mrakodrapů, které jsou složeny z více menších částí, které mohou mít stěny velmi blízko sebe, viz obrázek 5.7. Občas tento jev nastane i u obyčejných domů ve velké dálce, které jsou u sebe v těsné blízkosti.



Obrázek 5.7: Ukázka vzniklého artefaktu z-fight, kde jsou dvě textury moc blízko sebe a navzájem se snaží překrýt.

## 5.4 Pohyb kamery

Během tvorby intra byl pohyb kamery ovládán uživatelem, tzn. pomocí klávesnice a myši, a to z důvodu jednoduššího pohybu po scéně a snadnějšího hledání chyb. Výsledné intro však musí být neinteraktivní, proto musíme trajektorii a směr pohledu kamery uložit do programu. Intro je rozděleno do několika částí, které můžeme nazvat průlety. Ve většině průletů je trajektorie kamery dána Catmull-Rom křivkou, která je popsána v kapitole 2.9. V některých průletech se kamera pohybuje po kružnici, k čemuž nepotřebujeme Catmull-Rom křivku, vystačíme si pouze s goniometrickými funkcemi sinus a cosinus.

Směr pohledu kamery se v intru určuje několika způsoby. První možností je pevný bod ve scéně, na který je kamera pořád zaměřená. Druhou možností jsou dva body, mezi kterými interpolujeme a podle toho nastavíme pohled kamery. Třetí možností je, že se díváme na Catmull-Rom křivce „kousek před sebe“, jinak řečeno si spočítáme pozici bodu na trajektorii kamery, který je těsně před námi, a na ten zaměříme kameru.

Po vypočítání pozice a směru kamery můžeme spočítat tzv. *view matici*, což je matice, kterou transformujeme geometrii objektů a tím je zasadíme do souřadnicového systému kamery. K tomu se nám hodí funkce `lookAt` z knihovny GLM, která k výpočtu potřebuje pozici kamery a dva vektory, které jednoznačně udávají směr a natočení kamery v prostoru.

## 5.5 Hudba

Nedílnou součástí grafického intra je hudba, která dodává správnou atmosféru a zlepšuje celkový dojem výsledné prezentace. Kvůli omezené velikosti spustitelného souboru nelze použít obvyklé hudební formáty jako MP3, WAVE nebo FLAC, které obsahují audio data a zabírají hodně místa. Pro intra s omezenou velikostí se nabízí formát MIDI, který obsahuje seznam not a délku znění jednotlivých not. Jedná se tedy v podstatě o instrukce, které říkají přehrávači, jak má zvuk znít. Tento formát se kvůli jednoduchosti a minimální velikosti používá spíše pro grafická intra do 4kB.

V tomto intru je využit formát V2M, který oproti MIDI nabízí větší možnosti, vyšší kvalitu hudby a je vyvinut speciálně pro grafická intra do 64kB. Stejně jako u formátu MIDI se jedná o sérii instrukcí, k jejichž přehrání potřebujeme vhodný syntetizátor. Využit je V2 syntetizátor od skupiny Farbrausch, který je obsažen v knihovně libv2, viz kapitola 4.4. Protože se nejedná o často využívaný audio formát, není příliš velký výběr dostupných skladeb. Použita je skladba iOTA tune 2 [8] od neznámého autora.

## Kapitola 6

# Metody pro snížení velikosti aplikace

Jelikož je našim cílem vytvořit grafické intro do 64kB, musíme použít několik technik, které nám zmenší velikost spustitelného souboru. Tato kapitola popisuje použité parametry při kompilaci programu a program pro kompresi spustitelného souboru.

### 6.1 Nastavení překladače

Visual Studio nabízí velké množství parametrů, které mohou být použity při kompilaci pro optimalizaci výsledného kódu. Je důležité být ve Visual Studiu v tzv. *Release* módu, který neobsahuje žádné symboly a informace pro debugování. Následně můžeme přidat parametry pro optimalizaci, v tomto intru jsou to tyto parametry:

- `/QIfist` - pokud tento parametr není použit, tak kompilátor vloží volání funkce `_ftol()` při každém převodu z float na int, která je zodpovědná za správný převod podle IEEE standardu, což je pomalý proces a vytváří závislost na knihovně *libc*.
- `/GX-` - vypnutí zpracování výjimek
- `/O1` - použití tohoto parametru je ekvivalentní k použití parametrů `/Og /Os /Oy /Ob2 /Gs /GF /Gy` a vytvoří nejmenší možný spustitelný soubor
- `/Og` - globální optimalizace programu, eliminuje podvýrazy ve výpočtech, optimalizuje cykly a umožňuje automatickou alokaci často používaných proměnných v registrech
- `/Os` - minimalizuje velikost EXE a DLL souborů na úkor rychlosti programu
- `/Oy` - povoluje vynechání *frame pointeru*
- `/Ob2` - povoluje rozšíření inline funkcí
- `/Gs` - zapnutí kontroly zabezpečení vyrovnávací paměti
- `/GF` - odstraní duplicitní řetězce použité v kódu
- `/Gy` - povolení propojení na úrovni funkcí

- /GL - použití tohoto parametru provede celkovou optimalizaci programu s informacemi o všech modulech místo optimalizace na úrovni modulů.

Tato tabulka ukazuje velikost výsledného souboru po použití jednotlivých parametrů:

Parametr	Velikost s přidáním parametru
-	291kB
/QIfist	291kB
/GX-	290kB
/O1	274kB
/GL	273kB

Tabulka 6.1: Velikost spustitelného souboru s použitím jednotlivých parametrů.

## 6.2 Exe packer

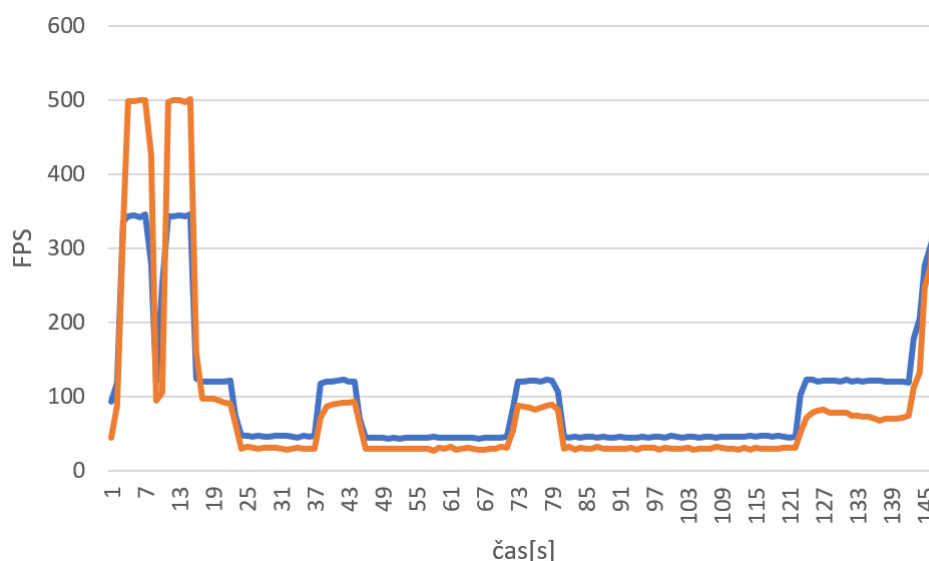
Jelikož se po použití všech parametrů, které zmenšují velikost výsledného souboru, ani zdaleka neblížíme k požadované velikosti spustitelného souboru, je potřeba využít tzv. *exe packer*. Tyto exe packery mají za úkol kompresi spustitelného souboru, což znamená jeho zmenšení se zachováním informací, které jsou v něm uloženy. Po spuštění takového souboru se znovuvytvoří originální kód, který je následně spuštěn. Toto může lehce zpomalit start programu.

Existuje mnoho exe packerů, například UPX, Petite, MPRESS a další. V této práci je využit exe packer kkrunchy od skupiny Farbrausch, který je přímo vytvořen pro kompresi grafických inter do 64kB. Po jeho použití se velikost spustitelného souboru zmenšila z 273kB na 61kB. Tento exe packer je ke stažení z [\[2\]](#).

## Kapitola 7

# Vyhodnocení

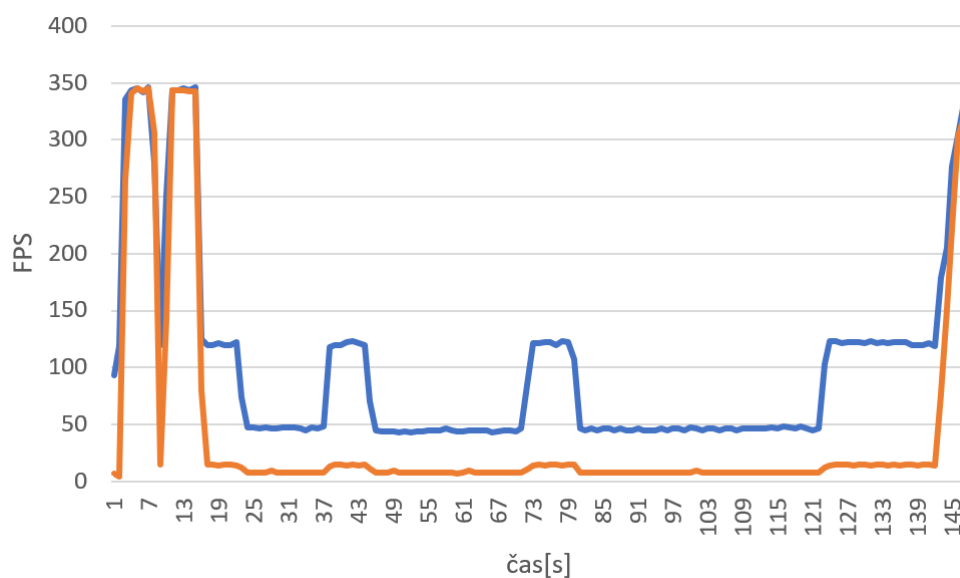
Grafické intro bylo vyvinuto na notebooku ASUS VivoBook 17, který disponuje grafickou kartou GeForce 920MX a procesorem Intel® Core™ i5-8250U. Intro lze spustit jak na integrované grafické kartě, tak na dedikované grafické kartě. Pokud však vykresluje s volumetrickými paprsky, probíhá na grafické kartě více výpočtů a hodnota *FPS* (počet snímků za sekundu) je na integrované grafické kartě nízká, proto je lepší použít dedikovanou grafickou kartu. Porovnání můžeme vidět v grafu 7.1.



Graf 7.1: Porovnání hodnot FPS při použití integrované grafické karty (oranžová barva) a dedikované grafické karty (modrá barva). Dva velké skoky na začátku videa, ve kterých je integrovaná grafická karta rychlejší, jsou kvůli černým pasážím, kde se nevyskresluje budovy, všude jinde dosahuje lepší výkonosti dedikovaná grafická karta.

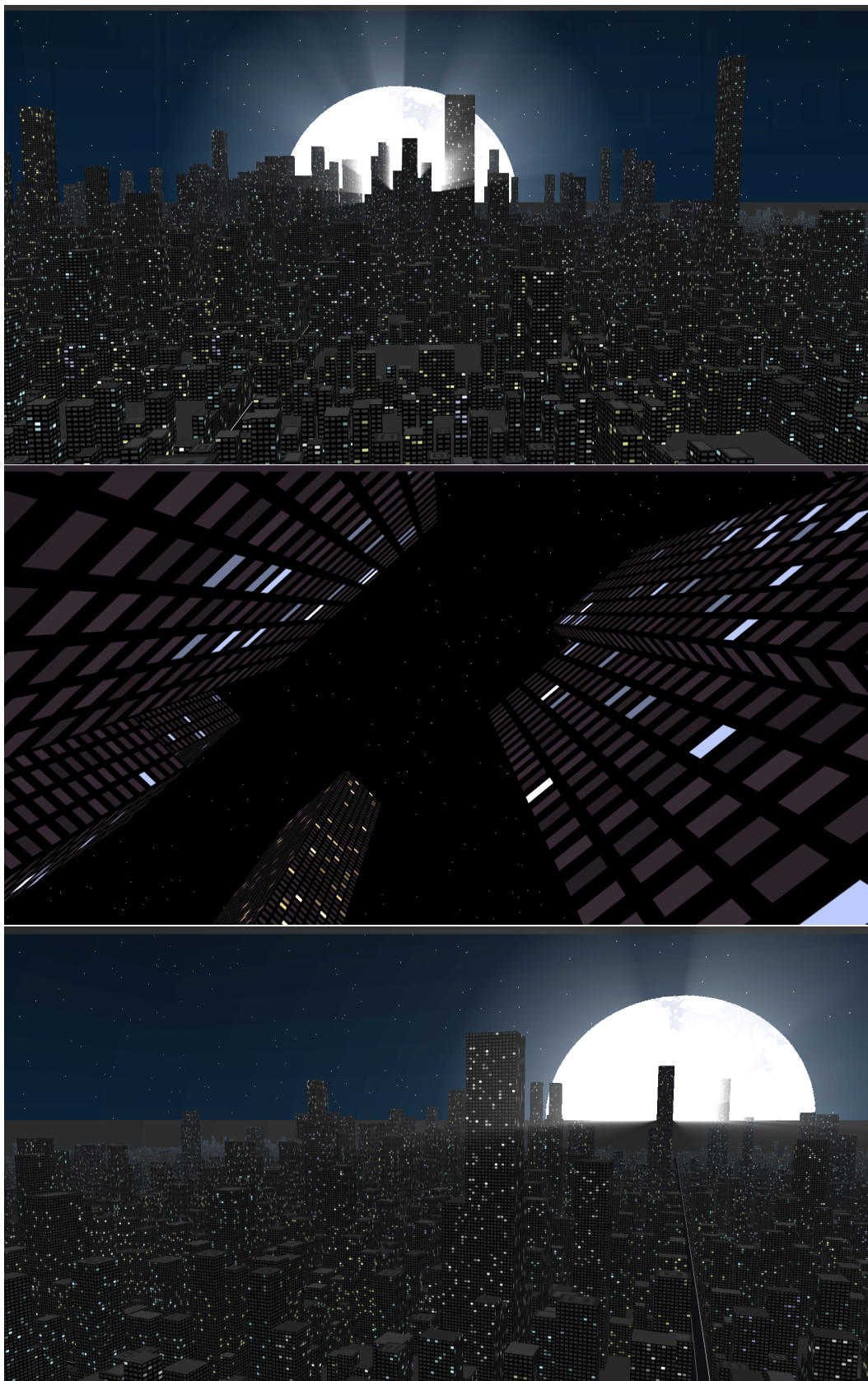
Po spuštění programu probíhá generování budov, silnic, textur a veškerého grafického obsahu, které trvá přibližně pět sekund. V intru se nachází 1089 bloků, ve kterých je 45621 budov. Využití CPU při běhu programu je přibližně 15%, využití paměti procesoru dosáhne během generování 987MB, při běhu programu po vygenerování všech objektů je využití

paměti 596MB. V grafu 7.2 můžeme vidět porovnání FPS při vykreslování snímků před a po použití instancingu, který značně urychlil vykreslování.



Graf 7.2: Porovnání hodnot FPS při použití instancingu (modrá barva) a bez jeho použití (oranžová barva). Rozdíly v počtu vykreslených snímků za sekundu jsou markantní.





Obrázek 7.3: Několik snímků výsledné scény.

## Kapitola 8

# Závěr

Cílem této bakalářské práce bylo vytvoření intra s omezenou velikostí do 64kB s použitím OpenGL. Jelikož jsem s knihovnou OpenGL neměl žádné praktické zkušenosti, bylo potřeba tuto knihovnu a její nadstavby nastudovat. Dále bylo potřeba nastudovat a implementovat techniky pro generování grafického obsahu a metody pro snížení výsledné velikosti spustitelného souboru.

Jako téma grafického intra jsem zvolil průlet nočním městem. V pozadí je velký měsíc, ze kterého vychází volumetrické paprsky. Mezi použité techniky patří

- procedurální generování (vytváření grafického obsahu)
- Perlinův šum (textura měsíce)
- mipmapping (odstranění aliasingu u textur)
- volumetrické paprsky (měsíční paprsky)
- face culling (omezení počtu vykreslovaných fragmentů)
- instancing (zrychlení vykreslování podobných objektů)
- mlha (dodání informace o vzdálenosti objektů)
- skybox (vytvoření pozadí scény)
- Catmull-Rom spline (křivka, po které se pohybuje kamera).

Výsledný spustitelný soubor má 61kB, což znamená, že podmínka omezené velikosti do 64kB je splněna. Délka výsledného intra je 147 sekund. V projektu lze pokračovat mnoha způsoby. Do intra by se dalo přidat pouliční osvětlení, auta projíždějící ulicemi, parky se zelení nebo změna počasí. Dále by se daly vylepšit texturu budov, případně by se dalo vytvořit více modelů budov. Jelikož je výsledná velikost těsně pod hranicí, muselo by se najít místo pro implementaci rozšíření, případně by se musela zvýšit hranice velikosti intra.

# Literatura

- [1] Catmull-Rom splines, Robotics, Teaching & Learning. [online]. [cit. 23.04.2018].  
URL <https://www.lucidarme.me/catmull-rom-splines/>
- [2] GitHub - farbrausch/fr\_public: Farbrausch demo tools 2001-2011. *The world's leading software development platform · GitHub*. [online]. Copyright © 2018 [cit. 02.05.2018].  
URL [https://github.com/farbrausch/fr\\_public](https://github.com/farbrausch/fr_public)
- [3] Glad - Multi-Language GL/GLES/EGL/GLX/WGL Loader-Generator based on the official specs. [online]. [cit. 02.05.2018].  
URL <http://glad.dav1d.de/>
- [4] GLFW - An OpenGL library. *GLFW - An OpenGL library*. [online]. [cit. 02.05.2018].  
URL <http://www.glfw.org/>
- [5] God Rays? What's that? – Community Play 3D – Medium. *Medium – Read, write and share stories that matter* [online]. [cit. 23.04.2018].  
URL <https://medium.com/community-play-3d/god-rays-whats-that-5a67f26aeac2>
- [6] Grafická knihovna OpenGL (25): mipmapping - Root.cz. *Root.cz - informace nejen ze světa Linuxu*. [online]. Copyright © 1998 [cit. 01.05.2018].  
URL <https://www.root.cz/clanky/opengl-25-mipmapping/>
- [7] Learn OpenGL, extensive tutorial resource for learning Modern OpenGL. *Learn OpenGL, extensive tutorial resource for learning Modern OpenGL*. [online]. [cit. 23.04.2018].  
URL <https://learnopengl.com>
- [8] Mamont's open FTP Index. [online]. [cit. 13.05.2018].  
URL <http://www.mmnt.net/db/0/1/ftp.undergrund.net/users/Freefall/V2/Modules/Others/>
- [9] OpenGL - Drawing polygons. *OpenGL - Introduction*. [online]. [cit. 11.05.2018].  
URL <https://open.gl/drawing>
- [10] OpenGL Mathematics. *GLM*. [online]. [cit. 02.05.2018].  
URL <https://glm.g-truc.net/0.9.8/index.html>
- [11] Rendering Pipeline Overview - OpenGL Wiki. *The Khronos Group Inc.* [online]. [cit. 11.05.2018].  
URL [https://www.khronos.org/opengl/wiki/Rendering\\_Pipeline\\_Overview](https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview)

- [12] The Yasm Modular Assembler Project. *The Yasm Modular Assembler Project*. [online]. Copyright © 2018 [cit. 05.05.2018]. URL <http://yasm.tortall.net/>
- [13] Hoffman, N.; Preetham, A. J.: Graphics Programming Methods. kapitola Real-time Light-atmosphere Interactions for Outdoor Scenes, Rockland, MA, USA: Charles River Media, Inc., 2003, ISBN 1-58450-299-1, s. 337–352.
- [14] Nguyen, H.: *GPU Gems 3*. Addison-Wesley Professional, 2007, ISBN 9780321545428.
- [15] Shaker, N.; Togelius, J.; Nelson, M. J.: *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016.
- [16] Shreiner, D.; Sellers, G.; Kessenich, J.: *OpenGL® Programming Guide: The Official Guide to Learning OpenGL®, Version 4.5 with SPIR-V, Ninth Edition*. Addison-Wesley Professional, 2016, ISBN 9780134495514.
- [17] Shreiner, D.; The Khronos OpenGL ARB Working Group: *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1, Seventh Edition*. Addison-Wesley Professional, 2009, ISBN 9780321669292.
- [18] Stanley, K. O.; Bryant, B. D.; Miikkulainen, R.: Real-time Neuroevolution in the NERO Video Game. *Trans. Evol. Comp*, ročník 9, č. 6, Prosinec 2005: s. 653–668, ISSN 1089-778X, doi:10.1109/TEVC.2005.856210. URL <http://dx.doi.org/10.1109/TEVC.2005.856210>